**PAR Laboratory Assignment Lab 4: Divide and Conquer parallelism with OpenMP: Sorting**

Oscar Faixat Llanas

Xavier Algarra Torelló

Grup: PAR1212

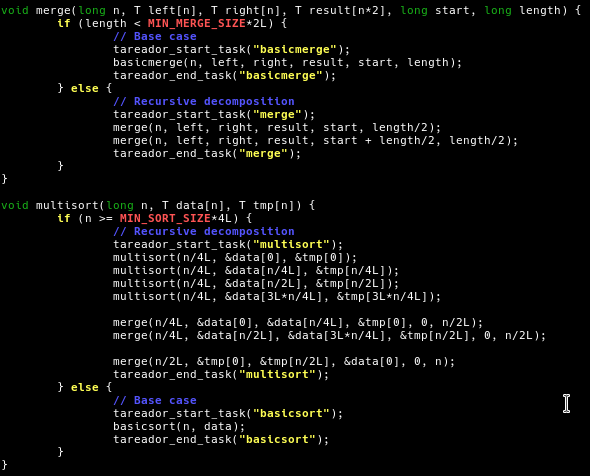
PAR 2017Q2

20/04/2017

# **4.1 Analysis with Tareador**

## **Include the relevant parts of the modified multisort-tareador.c code and comment where the calls to the Tareador API have been placed. Comment also about the task graph generated and the causes of the dependences that appear.**

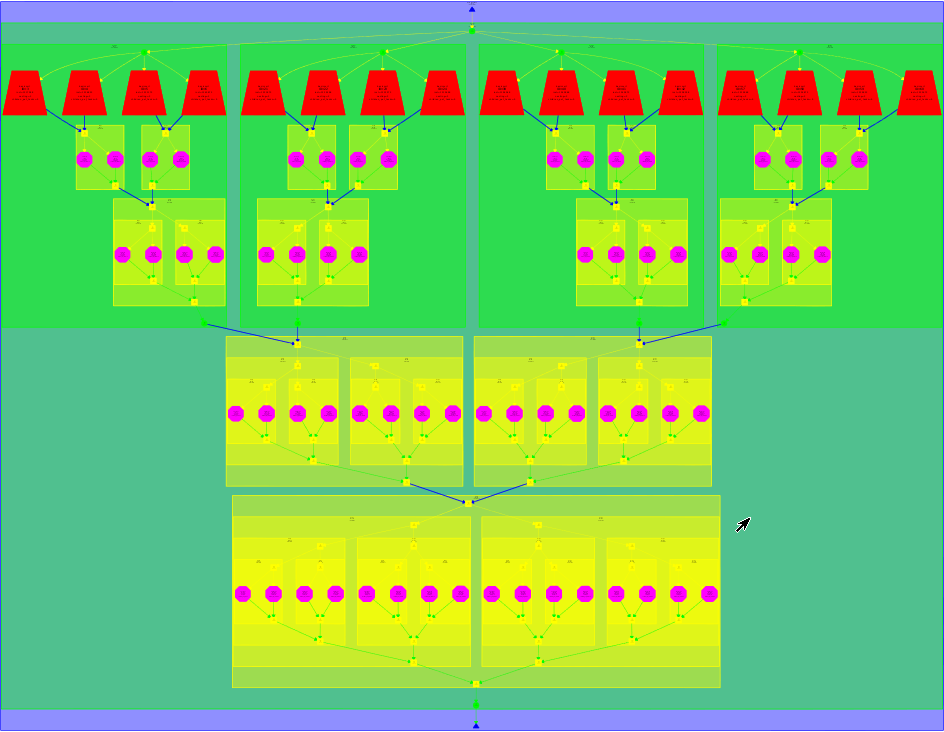
We have placed all of the calls to the tareador API enveloping the calls to the each of the functions basicmerge, merge, multisort and basicsort. The image below shows where we placed each call to the API.



When, the program starts it first enters the multisort function and divides the vector into 4 separate parts. Each part then gets executed again as a recursive call until it has the minimum size defined by MIN\_SORT\_SIZE\*4L. Once it is small enough, then the function basicsort is called to sort that portion of the vector. Once sorted, each portion is fused back together in pairs until the original size vector is reached (basically we undo the splitting we did recursively). Below we can observe the tareador task graph we have obtained using the code above.

## 

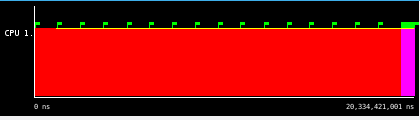
## 



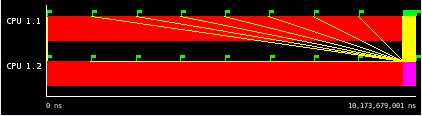
## 

## **Write a table with the execution time and speed-up predicted by Tareador (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer**

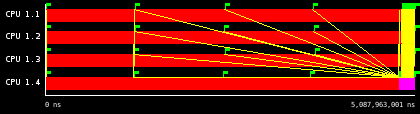
|  |  |  |
| --- | --- | --- |
|  | **Time (ns)** | **Speed Up** |
| **1 Processor** | 20,334,421,001 | - |
| **2 Processors** | 10,173,679,001 | 1.9987283852 |
| **4 Processors** | 5,087,963,001 | 3.9965740704 |
| **8 Processors** | 2,548,394,001 | 7.9793081419 |
| **16 Processors** | 1,289,949,001 | 15.76374026 |
| **32 Processors** | 1,289,949,001 | 15.76374026 |
| **64 Processors** | 1,289,949,001 | 15.76374026 |



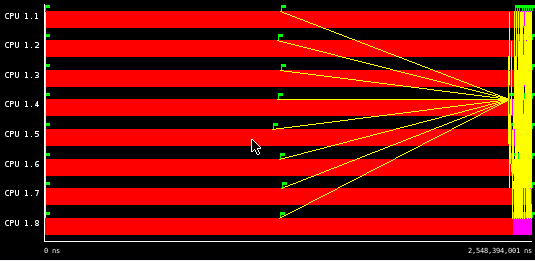
Trace with 1 processor.



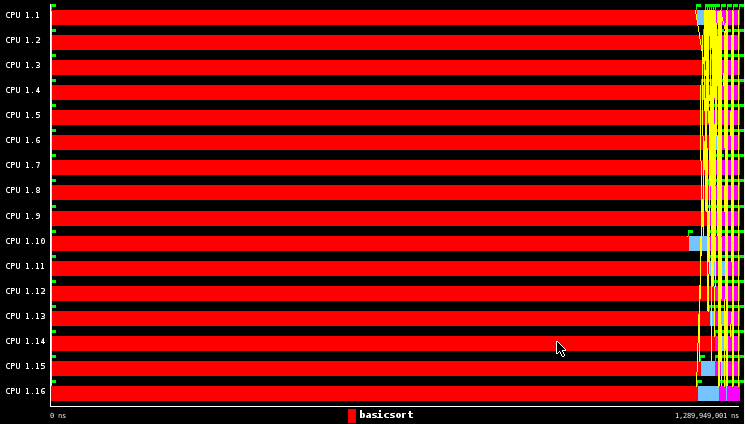
Trace with 2 processors.



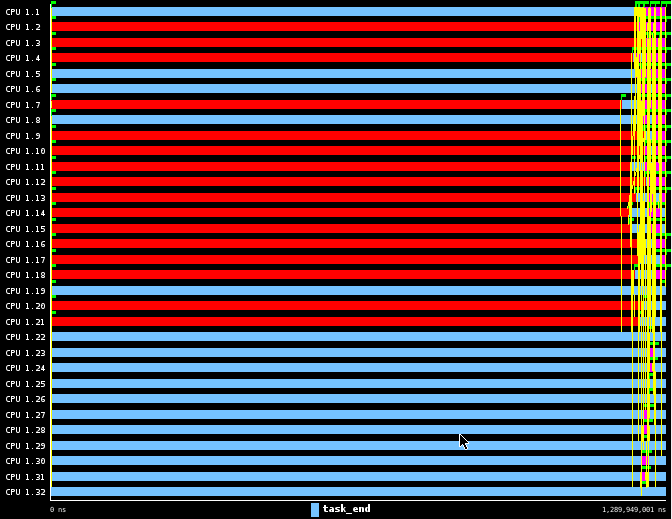
Trace with 4 processors.



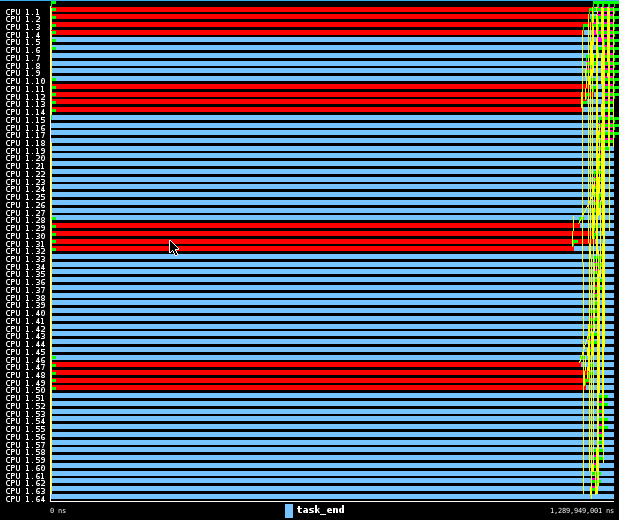
Trace with 8 processors.



Trace with 16 processors.



Trace with 32 processors.



Trace with 64 processors.

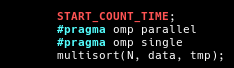
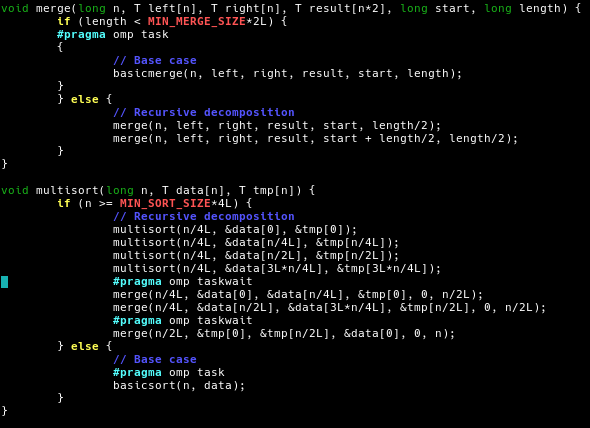
## The result we obtain from the Tareador simulation are close to the ideal case up to 16 cores. When using more than 16 cores, there is no time improvement. When looking at the traces of 32 and 64 cores above, we can see that a lot of cores are in the blue state (idle) instead of in the red state (working). Tareador is only using a maximum of 16 cores because the input data is too small to be able to be parallelised further. We have to use a small input data as otherwise tareador becomes too slow to be used. On a side note, we have to state that the overhead time is not taken into account as Tareador simulates the execution.

# 

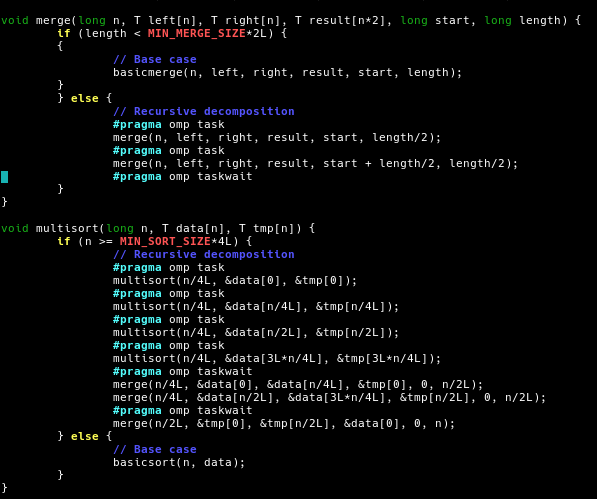
# **4.2 Parallelization and performance analysis with tasks**

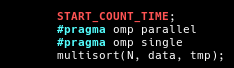
## **Include the relevant portion of the codes that implement the two versions (Leaf and Tree), commenting whatever necessary.**

For the Leaf parallelization strategy, we are using the pragma omp to indicate where we have the parallel regions and for thread creation. At the same, time though we are making use of the single pragma so that only one thread enters the multisort function. This way only that thread is in charge of creating the tasks for the merge and multisort base cases. The other threads stay idle until the previous thread is done creating and assigning tasks. In the multisort function we have placed two taskwaits because we detected that there were some task dependencies between the recursive calls of the merge functions and the calculations of the basicsort. This dependency occurs when both the merge and the basicsort access the same vector. To avoid it, we make the merge function wait until the sort has finished ordering the vector.



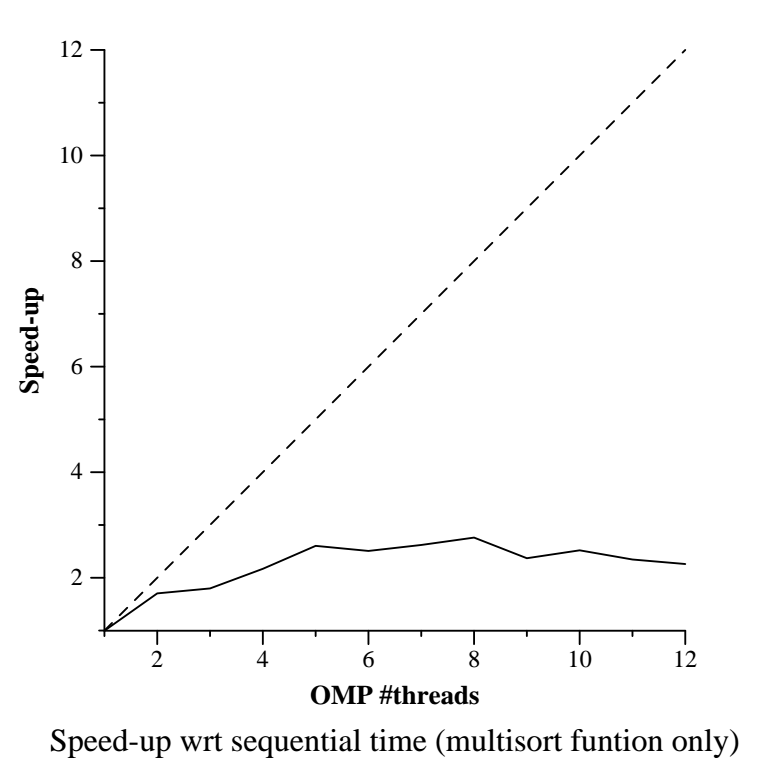
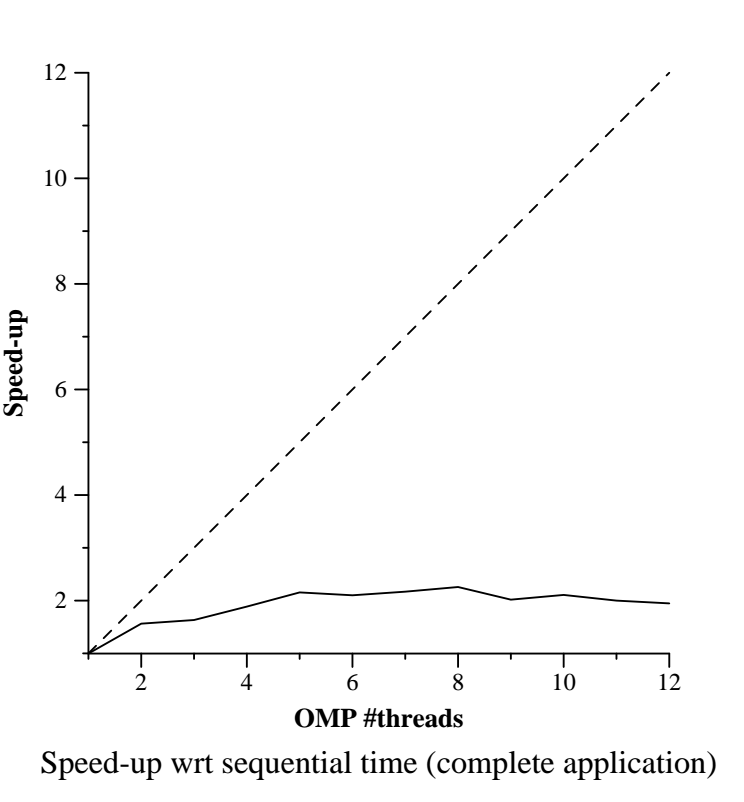
For the Tree parallelization, we are not creating tasks in the leaves of the tree’s recursive calls. What we do instead is, create them in the recursive decomposition without any cut-off levels. This will allow the thread to keep creating tasks until it reaches a leaf. While this is happening, all of the other threads will have to wait until the thread is finished creating tasks so that they can be assigned to tasks and execute them. In order to implement this, we are going to take advantage of the pragma taskwait to wait until the tasks are finished sorting their portion of vector before merging them. However, for this strategy we also have to apply the taskwait in the result vector as we need to wait for the merge function to end before being able to return to the level above in the tree.



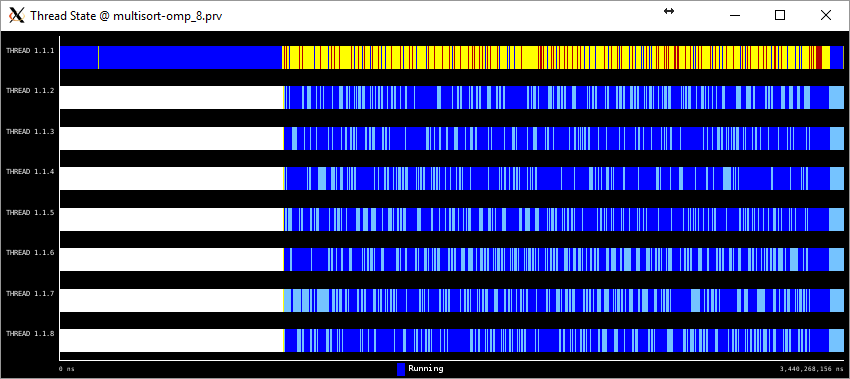


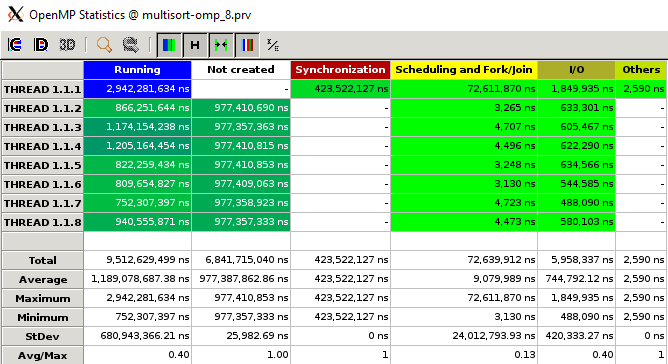
1. **For the the Leaf and Tree strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance that is observed, including captures of Paraver windows to justify your explanations.**

**Leaf Parallelization**

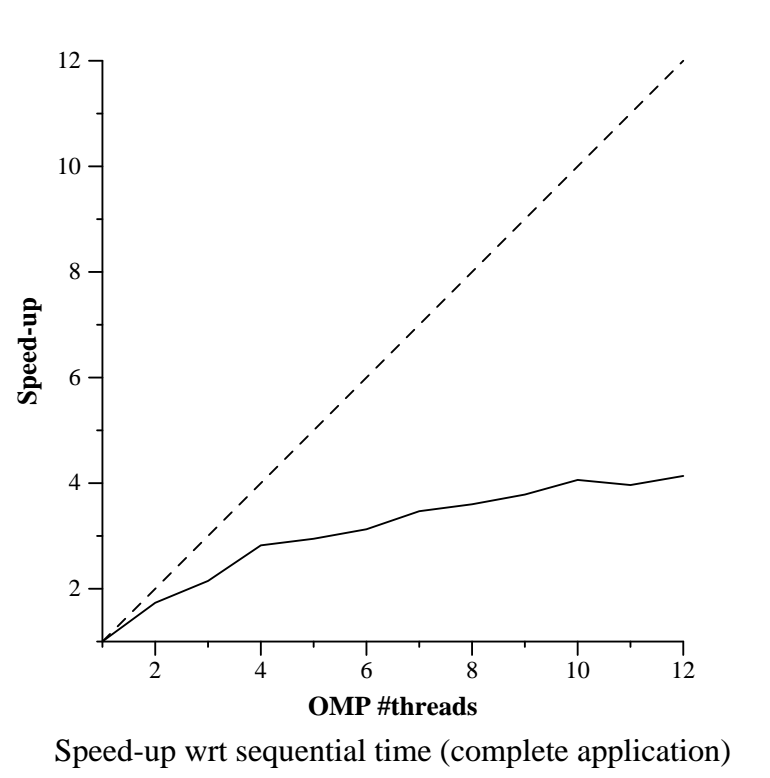
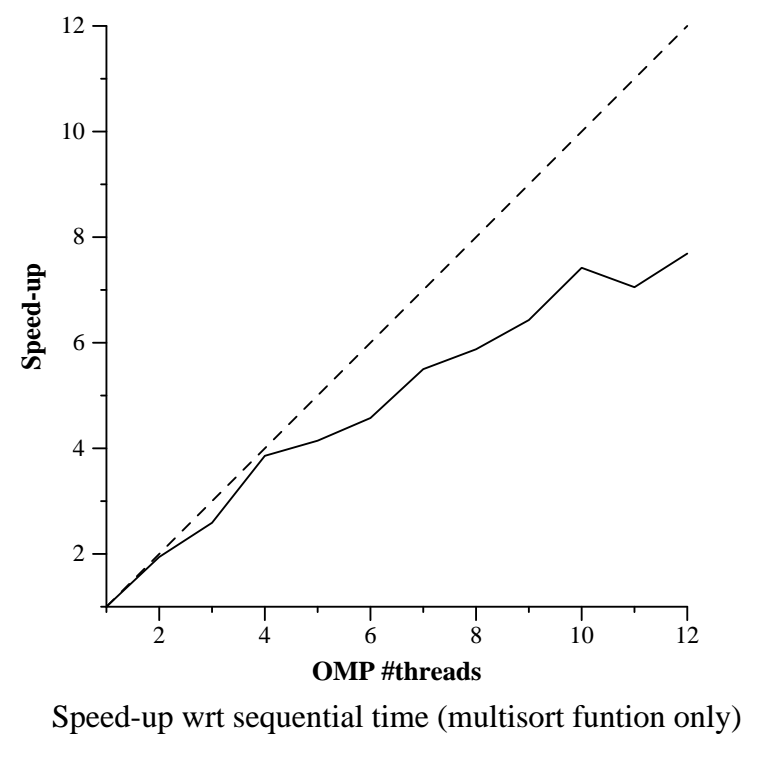
From the scalability plots above, we have the speedups of the whole program(left) and of the multisort function only (right). We can see that there is barely any benefits using this parallelization strategy as the speedup doesn’t even reach a value of three in both cases. From the timeline below we can observe that the program for the first third (aprox) of the execution it's only using one thread. It’s after this that further threads are created. However, even though more threads are being used, we can observe from both the timeline and statistics that the threads end up spending a decent amount of time idling (light blue) instead of working (dark blue). 

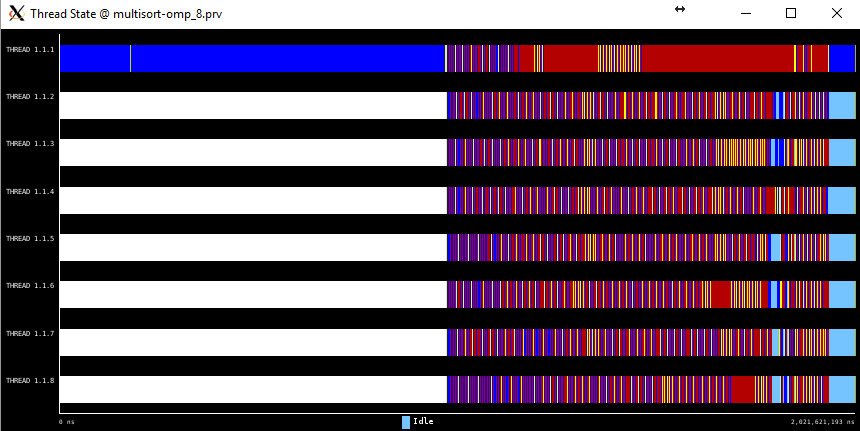
From all of the execution information we have gathered, the speedup in with this parallelization strategy is not better mainly because of the big chunk that is done sequentially. This sequential chunk therefore highly limits the potential speedup that can be obtained. However, that’s not the only reason for the limited speed-up, as the threads also spend quite a bit of time idling and as such we could infer that there is a bottleneck when creating tasks. We believe that this occurs because with the leaf parallelization strategy it takes the program quite a bit of time to reach the leaves of the recursive tree. Eventually, when the main thread does reach the leaves it does start creating the tasks and the other threads can start executing them.

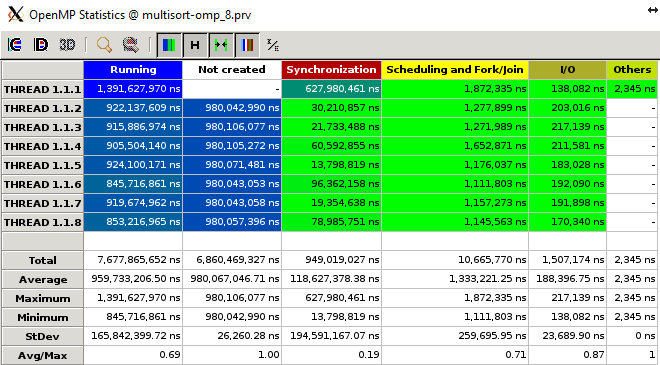




**Tree Parallelization**

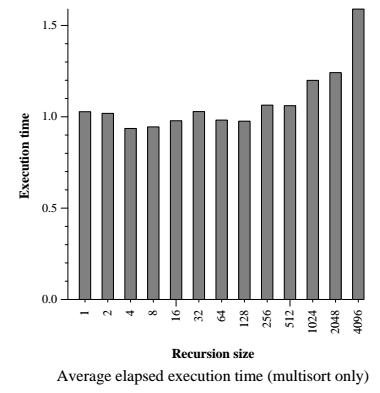


From the scalability plots above, we have the speedups of the whole program(left) and of the multisort function only (right). We can already see that the speedups compared to the leaf strategy are much better. In this case at least when the threads increase so does the speedup (even though not at a quick rate). We believe this is due to the fact that with this strategy all of the parallelization occurs in the recursive calls of the tree instead of in the leaves. Therefore, all of the threads that were previously idle no longer have to wait until a leaf is reached to create a task. From the timeline below we can observe that the threads barely spend any time idling. The drawback is that there is quite a big percentage of time spent due to the taskwait. 

It should be noted that the speed-up of only the multisort function is actually much greater than that of the whole program. This happens because as we can see from the timeline above almost half of the execution is done sequentially. This sequential execution is most probably given by the initialize and clear functions. As such, it limits the potential speed-up of the whole program. 

After comparing both strategies, we can conclude that for this program the Tree parallelization strategy is better than the Leaf parallelization strategy. We conclude this because there is a stronger scalability with tree than there is with leaf.

1. **Analyze the influence of the recursivity depth in the Tree version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?**

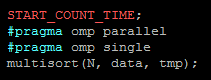
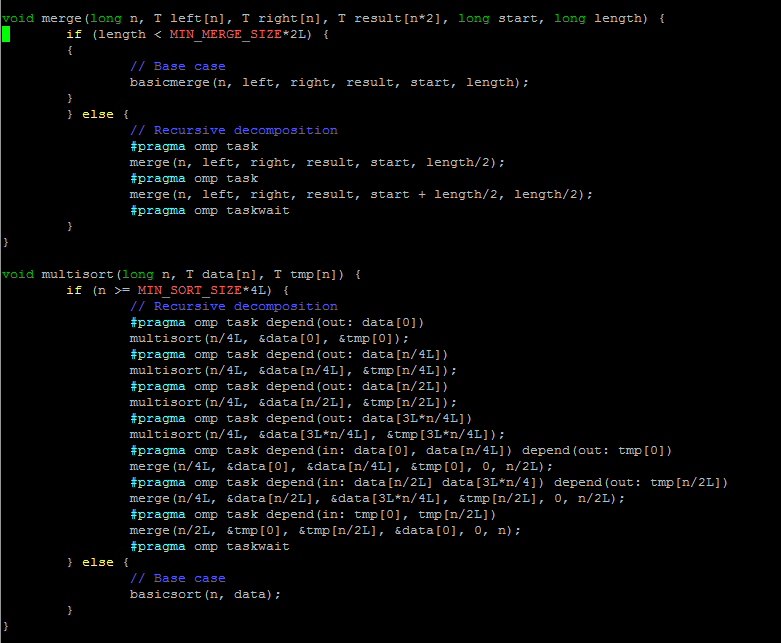


The image above, shows us that the bigger the recursion size, the higher the execution time. However, we can’t really obtain a valid conclusion as for example size 64 has a lower execution time than size 32. A possible explanation for this is that when we cut the tree depth with a higher value, we are losing parallelism and at the same we get less overhead because fewer tasks are created. Therefore, we are unable to confidently assure an optimal value.

The image above suggests that the bigger the recursion size the bigger the execution time will be. However, we can’t really obtain a valid conclusion because for example size 128 is faster than 32 and 16 is faster than 32. An explanation for this could be that when we cut the tree depth with a higher value we lose parallelism and at the same time get less overhead because we are creating fewer tasks. As such we can’t confidently say what the optimal value is.

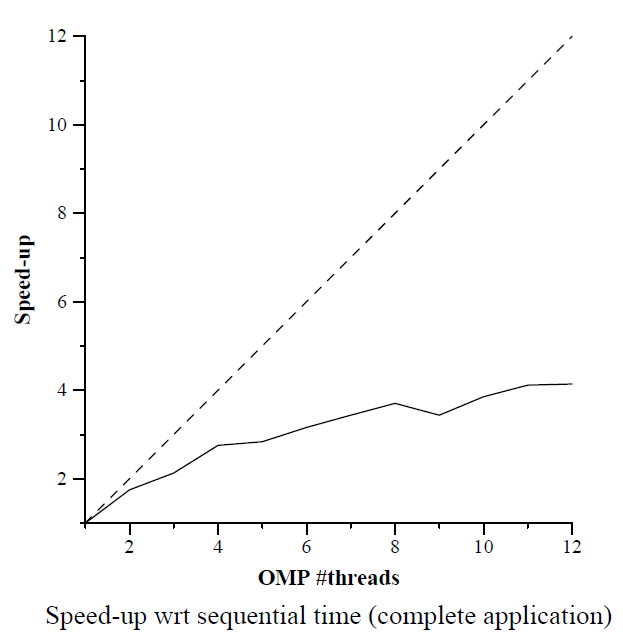
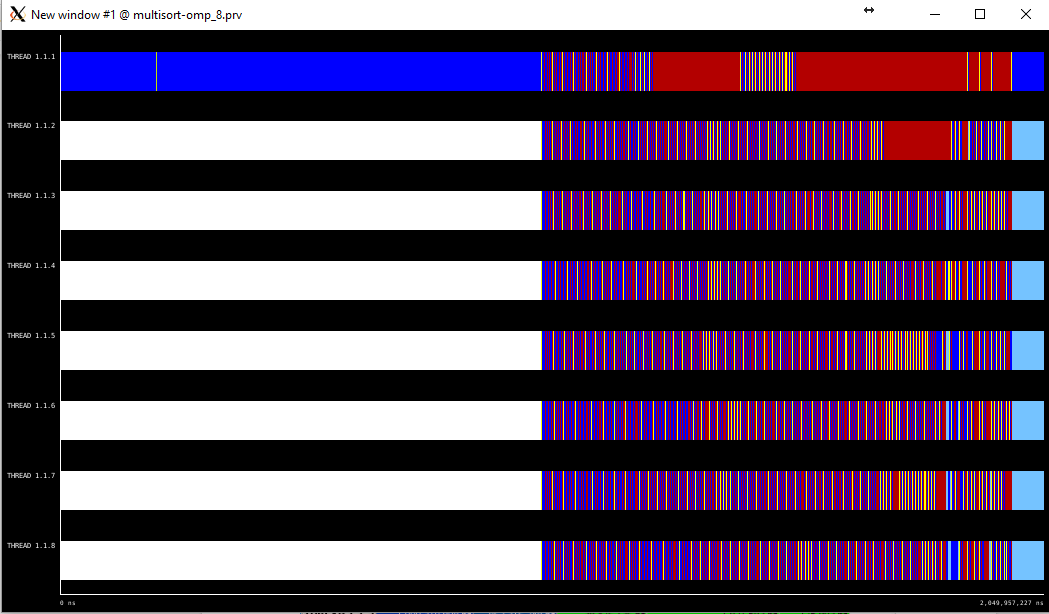
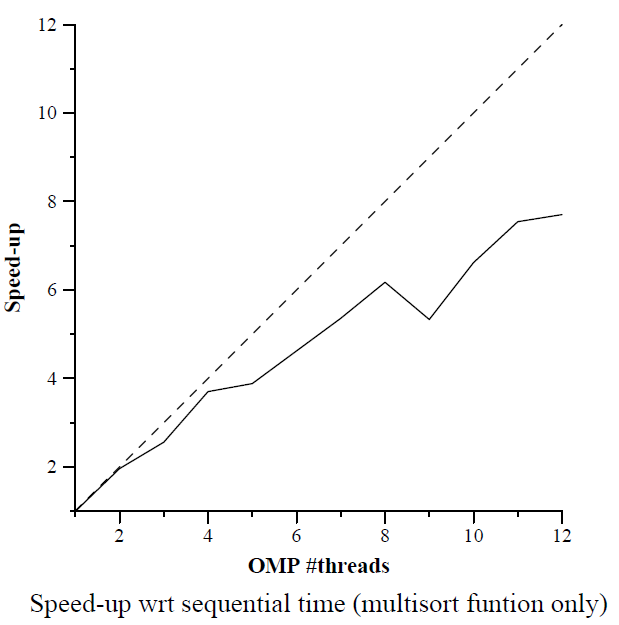
**4.3 Parallelization and performance analysis with dependent tasks**

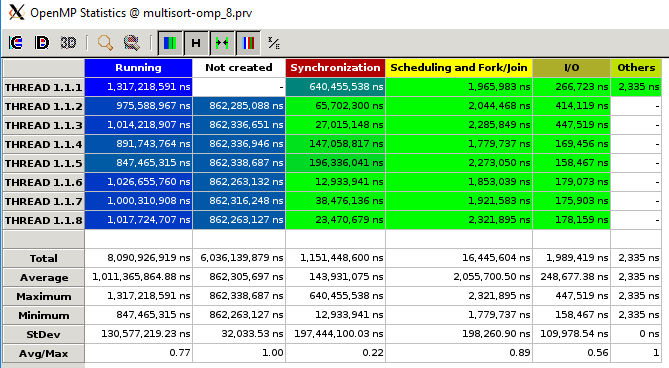
## **Include the relevant portion of the code that implements the Tree version with task dependencies, commenting whatever necessary.**



As we can see from the code above, the four recursive calls to the multisort function are executed with a pragma omp task that indicates that there are data dependencies of type out. This means that these tasks will only be considered valid if they are finished. Therefore if other tasks are waiting for them and these tasks are using dependencies of type in or out, the tasks that are waiting will have to wait until the dependencies are considered as valid. We find a similar case with the last merge function only this time we define the task as having type in dependencies. This merge function requires the in dependency because as it needs to receive both parts of tmp, the function won’t be able to be executed until both dependencies are met.

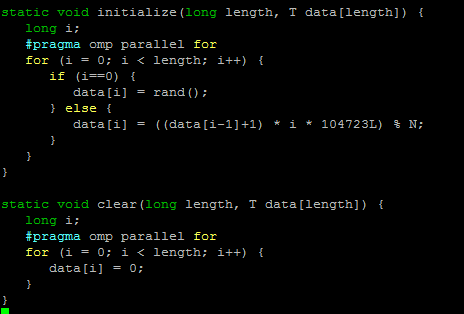
## **Reason about the performance that is observed, including the speed–up plots that have been obtained different numbers of processors and with captures of Paraver windows to justify your reasoning.**

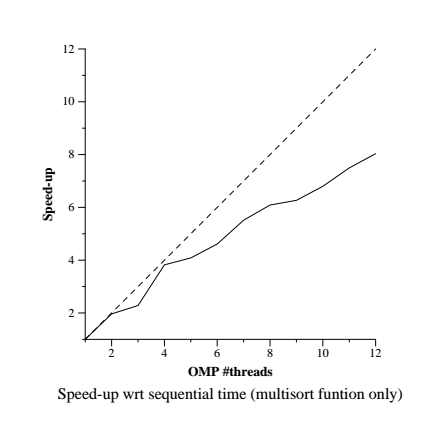
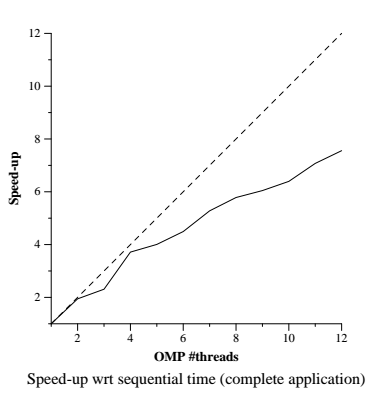




When comparing this execution with the previous execution of tree, we can see that overall there is barely any difference. However, in the execution of only the multisort function there are some differences. The main differences are that after 8 threads the speedup decreases and that there is a drop with 9 threads. From both the timeline and statistics we can see that there is a big amount of time being spent waiting (red) and a somewhat noticeable amount time scheduling (yellow). This is specially true for the main thread where it spends a big big chunk synchronizing. This is due to the fact that the program has to wait for the dependencies.

**Optional. If you have done any of the optional parts in this laboratory assignment, please include and comment in your report the relevant portions of the code and performance plots that have been obtained.**

**Optional 1:** Complete the parallelization of the Tree version by parallelizing the two functions that initialize the data and tmp vectors1 . Analyze the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the submit-strong-omp.sh script. Reason about the new performance obtained with support of Paraver timelines.



As we can observe from the speedups above, the timeline below and the statistics below parallelizing the initialization of the vector data really helps with the overall speedup of the program. Due to the changes we made, the program has a much smaller sequential chunk and this is what greatly benefits the overall speedup of the program. The multisort function speedup is pretty much the same as the changes we made doesn’t affect it. The program still has to wait for the taskwaits which is why there are still those big red chunks. In conclusion, the program overall spends less time executing sequentially and more parallelization can occur, therefore this change is highly recommended.

